Kernels on Prolog Proof Trees: Statistical Learning in the ILP Setting

Journal of Machine Learning Research 7 (2006) 307–342 Submitted 7/05; Revised 1/06; Published 2/06

Kernels on Prolog Proof Trees: Statistical Learning in the ILP Setting*

Andrea Passerini PASSERINI@DSI.UNIFI.IT
Paolo Frasconi P-F@DSI.UNIFI.IT
Dipartimento di Sistemi e Informatica
Università degli Studi di Firenze
Via di Santa Marta 3
I-50139 Firenze, Italy

Luc De Raedt DERAEDT@INFORMATIK.UNI-FREIBURG.DE
Institute for Computer Science
Albert-Ludwigs Universität Freiburg
Georges-Koehler-Allee 79
D-79110 Freiburg, Germany

Editor: Roland Olsson

Abstract

We develop kernels for measuring the similarity between relational instances using background knowledge expressed in first-order logic. The method allows us to bridge the gap between traditional inductive logic programming (ILP) representations and statistical approaches to supervised learning. Logic programs are first used to generate proofs of given visitor programs that use predicates declared in the available background knowledge. A kernel is then defined over pairs of proof trees. The method can be used for supervised learning tasks and is suitable for classification as well as regression. We report positive empirical results on Bongard-like and M-of-N problems that are difficult or impossible to solve with traditional ILP techniques, as well as on real bioinformatics and chemoinformatics data sets.

Keywords: kernel methods, inductive logic programming, Prolog, learning from program traces

1. Introduction

Within the fields of automated program synthesis, inductive logic programming (ILP) and machine learning, several approaches exist that learn from example-traces. An example-trace is a sequence of steps taken by a program on a particular example input. For instance, Biermann and Krishnaswamy (1976) have sketched how to induce Turing machines from example-traces (in this case sequences of primitive actions and assertions). Mitchell et al. (1983) have developed the LEX system that learned how to solve symbolic integration problems by analyzing traces (or search trees) for particular example problems. Ehud Shapiro's Model Inference System (1983) inductively infers logic programs by reconstructing

*. An early version of this paper was presented at the ICML '05 Workshop on Approaches and Applications of Inductive Programming (AAIP).

the proof-trees and traces corresponding to particular facts. Zelle and Mooney (1993) show how to speed-up the execution of logic programs by analyzing example-traces of the underlying logic program. Finally, De Raedt et al. (2005) proposed a method for learning stochastic logic programs using proof trees as training examples. The diversity of these applications as well as the difficulty of the learning tasks considered illustrate the power of learning from example-traces for a wide range of applications.

In this paper, we generalize the idea of learning from example-traces. Rather than explicitly learning a target program from positive and negative example-traces, we assume that a particular—so-called *visitor* program—is given and that our task consists of learning from the associated traces. The advantage is that in principle any programming language can be used to model the visitor program and that any machine learning system able to use traces as an intermediate representation can be employed. In particular, this allows us to combine two frequently employed frameworks within the field of machine learning: ILP and kernel methods. Logic programs will be used to generate traces corresponding to specific examples and kernels to quantify the similarity between traces. This combination yields an appealing and expressive framework for tackling complex learning tasks involving structured data in a natural manner. We call *trace kernels* the resulting broad family of kernel functions obtainable as a result of this combination. The visitor program is a set of clauses that can be seen as the *interface* between the available background knowledge and the kernel itself. Intuitively, the visitor program plays a role that is similar to that of declarative bias in inductive logic programming systems (Nédellec et al., 1996) (see also Section 6).

Kernels methods have been widely used in many relational learning contexts. Starting from the seminal work of Haussler (1999) (briefly reviewed in Section 4.1) several researchers have proposed kernels over discrete data structures such as sequences (Lodhi et al., 2002; Jaakkola and Haussler, 1999; Leslie et al., 2002; Cortes et al., 2004), trees (Collins and Duffy, 2002; Viswanathan and Smola, 2003), annotated graphs (Gärtner, 2003; Schölkopf and Warmuth, 2003; Kashima et al., 2003; Mahé et al., 2004; Horváth et al., 2004; Menchetti et al., 2005), and complex individuals defined using higher order logic abstractions (Gärtner et al., 2004). Constructing kernels over structured data types, however, is not the only aim of the proposed framework. In many symbolic approaches to learning, logic programs allow us to define background knowledge in a natural way. Similarly, in the case of kernel methods, the notion of similarity between two instances expressed by the kernel function is the main tool for exploiting the available domain knowledge. It seems therefore natural to seek a link between logic programs and kernels, also as a means for embedding knowledge into statistical learning algorithms in a *principled* and *flexible* way. This aspect is one of the main contributions of this paper as few alternatives exist to achieve this goal. Propositionalization, for example, transforms a relational problem into one that can be solved by an attribute-value learner by mapping data structures into a finite set of features (Kramer et al., 2000). Although it is known that in many practical applications propositionalization works well, its flexibility is generally limited. A remarkable exception is the method proposed by Cumby and Roth (2002) that uses description logic to specify features and that has been subsequently extended to specify kernels (Cumby and Roth, 2003). Muggleton et al. (2005) have proposed an approach where the feature space is spanned by a set of first order clauses induced by an ILP learning algorithm. Declarative kernels (Frasconi et al.,

2004) are another possible solution towards the above aim. A declarative kernel is essentially based on a background-knowledge dependent relation that allows us to extract *parts* from instances. Instances are reduced in this way to "bags-of-parts" and a combination of sub-kernels between parts is subsequently used to obtain the kernel between instances.

The guiding philosophy of trace kernels is very different from all the above approaches. Intuitively, rather than comparing two given instances directly, these kernels compare the execution traces of a program that takes instances as its input. Similar instances should produce similar traces when probed with programs that express background knowledge and examine characteristics they have in common. These characteristics can be more general than parts. Hence, trace kernels can be introduced with the aim of achieving a greater generality and flexibility with respect to various decomposition kernels (including declarative kernels). In particular, *any* program to be executed on data can be exploited within the present framework to form a valid kernel function, provided one can give a suitable definition of the *visitor* program to specify how to obtain relevant traces and proofs to compare examples. Although in this paper we only study trace kernels for logic programs, similar ideas could be used in the context of different programming paradigms and in conjunction with alternative models of computation such as finite state automata or Turing machines.

In this paper, we focus on a specific learning framework for Prolog programs. The execution trace of a Prolog program consists of a set of search trees associated with a given goal. To avoid feature explosion due to failed paths, which are typically much more numerous and less informative than successful ones, we resort to a reduced representation of traces based on proof trees (Russell and Norvig, 2002) that only maintain successful search paths. Proof trees can be conveniently represented as Prolog ground terms. Thus, in this case, kernels over traces reduce to Prolog ground terms kernels (PGTKs) (Passerini and Frasconi, 2005). These kernels (which are reviewed in Section 4.3) can be seen as a specialization to Prolog of the kernels between higher order logic individuals earlier introduced by Gärtner et al. (2004). Because of the special nature of terms in the present context, we also suggest some proper choices for comparing logical terms that represent proofs. One central advantage of the proposed method, as compared to inductive logic programming, is that it naturally applies to both classification and regression tasks.

The remainder of this paper is organized as follows. In Section 2 we review the traditional frameworks of statistical learning and ILP. In Section 3 we develop a new framework for statistical learning in the ILP setting and introduce visitor programs and their traces. In Section 4 we derive kernel functions over program traces represented as Prolog proof trees. In Section 5 we report an empirical evaluation of the methodology on some classic ILP benchmarks including Bongard problems, M-of-N problems on sequences, and real world problems in bioinformatics and chemoinformatics. Section 6 contains a discussion on the relations between our approach and traditional ILP methods, as well as explanation based learning (Mitchell et al., 1986). Finally, conclusions are drawn in Section 7.

2. Notation and Background

In this section, we briefly review some concepts related to supervised learning (from both the statistical and the ILP perspective) that will be used for defining the framework of learning from proof trees presented in the paper.

2.1 Statistical Learning and Kernels

In the usual statistical learning framework (see, e.g., Cucker and Smale, 2002, for a thorough mathematical foundation) a supervised learning algorithm is given a training set of input-output pairs $\mathcal{D} = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, with $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. The set $\mathcal{X}$ is called the input (or instance) space and can be any set. The set $\mathcal{Y}$ is called the output (or target) space; in the case of binary classification $\mathcal{Y} = \{-1, 1\}$ while the case of regression $\mathcal{Y}$ is the set of real numbers. A fixed (but unknown) probability distribution on $\mathcal{X} \times \mathcal{Y}$ links input objects to their output target values. The learning algorithm outputs a function $f : \mathcal{X} \mapsto \mathcal{Y}$ that approximates the probabilistic relation between inputs and outputs. The class of functions that is searched is called the *hypothesis space*.

A (Mercer) kernel is a positive semi-definite symmetric function[1] $K : \mathcal{X} \times \mathcal{X} \mapsto I\!R$ that generalizes the notion of inner product to arbitrary domains (see, e.g., Shawe-Taylor and Cristianini, 2004, for details). When using kernel methods in supervised learning, the hypothesis space, denoted $\mathcal{F}_K$, is the so-called reproducing kernel Hilbert space (RKHS) associated with K. Learning consists of solving the following Tikhonov regularized problem:

$$f = \arg\min_{h \in \mathcal{F}_K} C \sum_{i=1}^{m} V(y_i, h(x_i)) + \|h\|_K \tag{1}$$

where $V(y, h(x))$ is a positive function measuring the loss incurred in predicting $h(x)$ when the target is y, C is a positive regularization constant, and $\|\cdot\|_K$ is the norm in the RKHS. Popular algorithms in this framework include support vector machines (SVM) (Cortes and Vapnik, 1995) and kernel ridge regression (Poggio and Smale, 2003; Shawe-Taylor and Cristianini, 2004). The representer theorem (Kimeldorf and Wahba, 1970) shows that the solution to the above problem can be expressed as a linear combination of the kernel between individual training examples x_i and x as follows:

$$f(x) = \sum_{i=1}^{m} c_i K(x, x_i). \tag{2}$$

The above form also encompasses the solution found by other algorithms such as the kernel perceptron (Freund and Schapire, 1999).

2.2 Inductive Logic Programming

Within the field of inductive logic programming, the standard framework is that of learning from entailment. In this setting, the learner is given a set of positive and negative examples $\mathcal{D}^+$ and $\mathcal{D}^-$, respectively (in the form of ground facts), and a background theory $\mathcal{B}$ (as a set of definite clauses) and has to induce a hypothesis $\mathcal{H}$ (also a set of definite clauses) such that $\mathcal{B} \cup \mathcal{H}$ covers all positive examples and none of the negative ones. More formally, $\forall p(x) \in \mathcal{D}^+ : \mathcal{B} \cup \mathcal{H} \models p(x)$ and $\forall p(x) \in \mathcal{D}^- : \mathcal{B} \cup \mathcal{H} \not\models p(x)$. In this paper, as in the work by Lloyd (2003), we shall use examples that are individuals, i.e., first-order logic objects or identifiers. This means that we shall effectively refer to the examples by their identifier x rather than use $p(x)$. The traditional definition of inductive logic programming does not

1. A symmetric function $K : \mathcal{X} \times \mathcal{X} \mapsto I\!R$ is called a *positive semi-definite kernel* iff $\forall\, m \in I\!N, \forall\, x_1, \ldots, x_m \in \mathcal{X}, \forall\, a_1, \ldots, a_m \in I\!R$, $\sum_{i,j=1}^{m} a_i a_j K(x_i, x_j) \geq 0$.

```
mutagenic(d26).

lumo(d26, -2.072).
logp(d26, 2.17).
atm(d26,d26_1,c,22,-0.093).
atm(d26,d26_2,c,22,-0.093).
atm(d26,d26_3,c,22,-0.093).
atm(d26,d26_4,c,22,-0.093).
atm(d26,d26_5,c,22,-0.093).
atm(d26,d26_6,c,22,-0.093).
atm(d26,d26_7,h,3,0.167).
atm(d26,d26_8,h,3,0.167).
atm(d26,d26_9,h,3,0.167).
atm(d26,d26_10,cl,93,-0.163).
atm(d26,d26_11,n,38,0.836).
atm(d26,d26_12,n,38,0.836).
atm(d26,d26_13,o,40,-0.363).
atm(d26,d26_14,o,40,-0.363).
atm(d26,d26_15,o,40,-0.363).
atm(d26,d26_16,o,40,-0.363).
bond(d26,d26_1,d26_2,7).
bond(d26,d26_2,d26_3,7).
bond(d26,d26_3,d26_4,7).
bond(d26,d26_4,d26_5,7).
bond(d26,d26_5,d26_6,7).
bond(d26,d26_6,d26_1,7).
bond(d26,d26_1,d26_7,1).
bond(d26,d26_3,d26_8,1).
bond(d26,d26_6,d26_9,1).
bond(d26,d26_10,d26_5,1).
bond(d26,d26_4,d26_11,1).
bond(d26,d26_2,d26_12,1).
bond(d26,d26_13,d26_11,2).
bond(d26,d26_11,d26_14,2).
bond(d26,d26_15,d26_12,2).
bond(d26,d26_12,d26_16,2).
```

```
nitro(X,[Atom0,Atom1,Atom2,Atom3]) :-
    atm(X,Atom1,n,38,_),
    bondd(X,Atom0,Atom1,1),
    bondd(X,Atom1,Atom2,2),
    atm(X,Atom2,o,40,_),
    bondd(X,Atom1,Atom3,2),
    Atom3 @> Atom2,
    atm(X,Atom3,o,40,_).

bondd(X,Atom1,Atom2,Type) :-
    bond(X,Atom1,Atom2,Type).
bondd(X,Atom1,Atom2,Type) :-
    bond(X,Atom2,Atom1,Type).
```

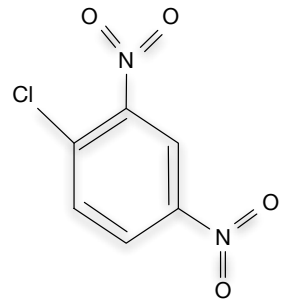

Figure 1: Example from the mutagenesis domain. Top: *extensional* representation of an instance (a molecule). Left: sample fragment of *intensional* background theory. Right: chemical structure of the molecule.

explicitly—as is the case of regularized empirical risk minimization—account for noisy data and the possibility that a complete and consistent hypothesis might not exist. Even though various noise handling techniques exist in inductive logic programming, they are not as principled as those offered by statistical learning theory.

Example 1 *As an illustration of the above concepts, consider the famous mutagenicity benchmark by Srinivasan et al. (1996). There the examples are of the form* `mutagenic(id)` *where* `id` *is a unique identifier of the molecule and the background knowledge contains information about the atoms, bonds and functional groups in the molecule. A hypothesis in this case could be*

```
mutagenic(ID)  ← nitro(ID,R),lumo(ID,L), L<-1.5.
```

It entails (covers) the molecule listed in Figure 1. It will be convenient to distinguish extensional *predicates, such as* `atm`, `logp`, `lumo` *and* `bond`, *which specify information about* specific *examples, from the* intensional *ones, such as* `bbond` *and* `nitro`, *which specify general properties about all examples.*

Regression can be introduced in ILP in different ways. For example in the First-Order Regression System (Karalič and Bratko, 1997) some arguments of the target predicate (called continuous attributes) are real-valued. For instance, in our example one could use examples of the form `mutagenic(d26, -2.072, 2.17, 6.3)` where the arguments would be the lumo and logp values as well as the target activity. FORS then learns from "positive" examples only, covering subsets of examples on which linear regression between the continuous arguments is solved in a numerical way. An interesting alternative is Structural Regression Trees, a method based on divide-and-conquer, similar to regression trees (Kramer, 1996).

3. A Framework for Statistical Learning in the ILP Setting

In this section we introduce the logical framework for defining program traces and, in particular, the concepts of visitor programs and proof trees.

3.1 General Assumptions

The methods described in this paper are based on a framework that combines some of the advantages of the statistical and the ILP settings, in particular noise robustness and the possibility of describing background knowledge in a flexible declarative language. First, we assume that the instance space $\mathcal{X}$ is a set of first-order logic objects (i.e., individuals in the universe of discourse), each having a unique identifier x. As in the ILP setting, we assume that a background theory $\mathcal{B}$ is available in the form of a set of definite clauses. This background theory is divided into *intensional* predicates, which are relevant to all examples, and *extensional* ones, which specify facts about specific examples. As in the statistical setting, we assume that a fixed and unknown distribution is defined on $\mathcal{X} \times \mathcal{Y}$ and that training data $\mathcal{D}$ consist of input-output pairs (x_i, y_i) (for classification or regression). Rather than having to find a set of clauses $\mathcal{H}$, the learning algorithm outputs a function f that maps instances into their targets and whose general form is given by Equation (2). In this sense, our setting is close to statistical learning and predictions on new instances will be essentially opaque. However, we make the fundamental assumption that f also depends on the available background theory via the kernel function.

3.2 Visitors

A second key difference with respect to the traditional ILP setting is that in addition to data $\mathcal{D}$ and background knowledge $\mathcal{B}$, the learner is given an additional set of clauses forming the so-called *visitor* program. Clauses in this program should be designed to "inspect" examples using other predicates declared in $\mathcal{B}$. In facts, as detailed in Section 4, the kernel function to be plugged in Equation (2) will be defined by means of the trace of this program. To this aim, we are not only interested in determining whether certain clauses succeed or fail on a particular example. In our approach, the execution traces of the visitor programs are recorded and compared, on the rationale that examples having similar traces should be mapped to similar representations in the feature space associated with the kernel. The purpose of visitors is thus to construct useful features during their execution. This is a major

difference with respect to other approaches in which features are explicitly constructed by computing the truth value for predicates (Muggleton et al., 2005).

Definition 1 (Visitor programs) *A* visitor *program for a background theory $\mathcal{B}$ and domain $\mathcal{X}$ is a set $\mathcal{V}$ of definite clauses that contains at least one special clause (called a* visitor*) of the form $V \leftarrow B_1, \ldots, B_N$ and such that*

- *V is a predicate of arity 1*
- *for each $j = 1, \ldots, N$, B_j is declared in $\mathcal{B} \cup \mathcal{V}$;*

Intuitively, if `visit/1` is a visitor in $\mathcal{V}$, by answering the query `visit(x)?` we explore the features of the instance whose constant identifier `x` is passed to the visitor. Having multiple visitors in the program $\mathcal{V}$ allows us to explore different aspects of the examples and include multiple sources of information.

Some examples of visitor programs are introduced in the remainder of this section and when presenting empirical results in Section 5.

3.3 Traces and Proof Trees

A visitor program trace for a given domain instance is obtained by recording proofs of visitor goals called on that instance. There are alternative options for choosing the kind of proof to be employed. Therefore in order to give a precise definition of traces, we now need to make a specific design choice. In this paper, we are committed to Prolog-based representations. Hence, a natural option would be the use of SLD-trees, whose paths correspond to execution sequences of the Prolog interpreter. A drawback of this choice is that an SLD-tree is a very complex and rather unstructured representation and also contains information about failed paths, potentially leading to an explosion of redundant and irrelevant features for the purpose of learning. For these reasons we prefer to resort to proof trees (Russell and Norvig, 2002), defined as follows:

Definition 2 (Proof tree) [2] *Let $\mathcal{P}$ be a program and G a goal. If $\mathcal{P} \not\models G$ then the* proof tree *for G is empty. Otherwise, it is a tree t recursively defined as follows:*

- *if there is a fact f in $\mathcal{P}$ and a substitution θ such that $G\theta = f\theta$, then $G\theta$ is a leaf of t.*
- *otherwise there must be a clause $H \leftarrow B_1, ..., B_n \in \mathcal{P}$ and a substitution θ' such that $H\theta' = G\theta'$ and $\mathcal{P} \models B_j\theta'$ $\forall j$, $G\theta'$ is the root of t and there is a subtree of t for each $B_j\theta'$ which is a proof tree for $B_j\theta'$.*

The kernels used in this paper work on ground proof trees. In general, however, proof trees or SLD-trees need not be ground. If they are not, they can however always be made ground by skolemization, i.e., by substituting all variables by different constants not yet appearing in the program and goal. The skolemized proof will then still logically follow from the program. Alternatively, one could impose the requirement that all clauses are

2. Such trees are sometimes also named *and-trees*.

range-restricted, a requirement that is often imposed in the logic programming community. Range-restrictedness requires that all variables that appear in the head of a clause also appear in its body. It is a sufficient requirement for guaranteeing that all proofs will be ground. Finally, ground proofs can be also obtained by making specific assumptions about the mode of head variables not occurring in the body, so that these variables will be instantiated in proving the goal. All the visitor programs presented in our empirical evaluation (see Section 5) yield ground proofs thanks to such assumptions.

Example 2 *For the sake of illustration, consider again the mutagenesis domain. Consider the atom bond representation of the simple molecule in Figure 1. By looking at the molecule as a graph where atoms are nodes and bonds are edges, we can introduce the common notions of* path *and* cycle*:*

```
1 : cycle(X,A):-
      path(X,A,B,[A]),
      bond(X,B,A,_).

2 : path(X,A,B,M):-
      atm(X,A,_,_,_),
      bond(X,A,B,_),
      atm(X,B,_,_,_),
      not(member(B,M)).

3 : path(X,A,B,M):-
      atm(X,A,_,_,_),
      bond(X,A,C,_),
      not(member(C,M)),
      path(X,C,B,[C|M]).
```

The following simple visitor may be used to inspect cycles in the molecule:

```
4 : visit(X):
      cycle(X,A).
```

Note that we numbered each clause in $\mathcal{V}$ and the intensional part of the background theory $\mathcal{B}$ (but not in the extensional part[3]) with a unique identifier. This will allow us to take into account information about the clauses that are used in a proof. The corresponding proof tree for this example is shown in Figure 2.

In general, a goal can be satisfied in more than one way. Therefore, each query generates a (possibly empty) set of proof trees. Since multiple visitors may be available, the trace of an instance is a tuple of sets of proof trees, as formalized in the following definition:

Definition 3 (Trace) *Let N be the number of visitors in $\mathcal{V}$ and for each $l = 1, \ldots, N$ let $T_{lj,x}$ denote the proof tree that represents the j-th proof of the goal $V_l(x)$, i.e., a proof that $\mathcal{B} \cup \mathcal{V} \models V_l(x)$. Let*

$$T_{l,x} = \{T_{l1,x}, \ldots, T_{ls_{l,x},x}\} \tag{3}$$

where $s_{l,x} \geq 0$ is the number of alternative proofs of goal $V_l(x)$. The trace of an instance x is the tuple

$$T_x = [T_{1,x}, \ldots, T_{N,x}]. \tag{4}$$

3.4 Pruning Proof Trees

In many situations, the proof tree for a given goal will be unnecessary complex in that it may contain several uninteresting subtrees. In these cases, we will often work with *pruned* proof trees, which are trees where subtrees rooted at specific predicates (declared as `leaf` predicates by the user) are turned into leafs. This will reduce the complexity of

3. The numbers in the extensional part would change from example to example and hence, would not carry any useful information.

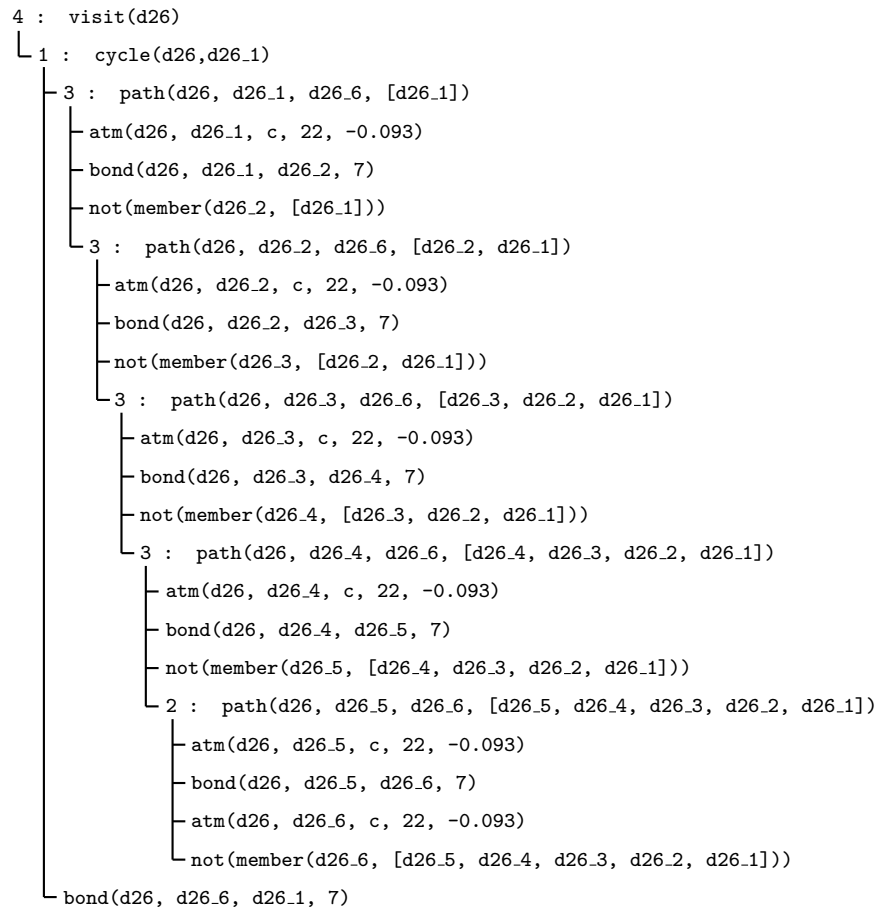

Figure 2: Proof tree resulting from the goal `visit(d26)` in the mutagenesis example.

the feature space associated with the kernel by selectively ignoring subproofs. For instance, consider again the mutagenesis domain described in Srinivasan et al. (1996) where a theory of rings and functional groups is included as background knowledge (see Figure 1). In this domain, it may be useful to define visitors that explore groups such as benzene rings:

```
atoms(X,[]).
atoms(X,[H|T]):-
    atm(X,H,_,_,_),
    atoms(X,T).
```

```
visit_benzene(X):-
    benzene(X,Atoms),
    atoms(X,Atoms).
```

If we believe that the presence of the ring and the nature of the involved atoms represent a sufficient set of features, we may want to ignore details about the proof of the predicate `benzene` by pruning the corresponding proof subtree. This can be accomplished by including the following fact in the visitor program:

```
leaf(benzene(_,_)).
```

3.5 Bridging the Gap

We are finally able to give a complete formalization of our framework for learning from example-traces. The learner is given a data set $\mathcal{D} = \{(x_1, y_1), \ldots, (x_m, y_m)\}$, background knowledge $\mathcal{B}$, and visitor program $\mathcal{V}$. For each instance x_i, a trace T_{x_i} is obtained by running the visitor program according to Definition 3. A kernel machine (e.g., an SVM) is then trained to form the function $f : \mathcal{X} \mapsto \mathcal{Y}$ defined as

$$f(x) = \sum_{i=1}^{m} c_i K(T_{x_i}, T_x).$$

The only missing ingredient is the kernel function K for comparing two visitor traces. The definition of this function is detailed in the next section.

4. Kernels over Visitor Traces

In this section, we derive kernel functions for comparing traces of visitor programs. We begin by reviewing some preliminary concepts about convolution kernels (Haussler, 1999), a very general family of kernels on discrete structures that will be used in the rest of the paper to define kernels over the logical structures of interest.

4.1 Kernels for Discrete Structures

For the purposes of this subsection, let $\mathcal{X}$ be a set of composite structures and for $x \in \mathcal{X}$ let $x_1, \ldots, x_D$ denote the "parts" of x, with $x_d \in \mathcal{X}_d$ for all $i \in [1, D]$. This decomposition can be formally represented by a relation R on $\mathcal{X}_1 \times \cdots \times \mathcal{X}_D \times \mathcal{X}$ such that $R(x_1, \ldots, x_D, x)$ is true iff $x_1, \ldots, x_D$ are the parts of x. We also write $(x_1, \ldots, x_D) = R^{-1}(x)$ if $R(x_1, \ldots, x_D, x)$ is true. Note that the relation R used in this definition is very general and does not necessarily satisfy an axiomatic theory for parts and wholes such as those studied in knowledge representation (Varzi, 1996). For example if $\mathcal{X}_1 = \cdots = \mathcal{X}_D = \mathcal{X}$ are sets containing all finite strings over a finite alphabet, we can define a relation $R(x_1, \ldots, x_D, x)$ which is true iff $x = x_1 \circ \cdots \circ x_D$, with $\circ$ denoting concatenation of strings. Note that in this example x can be decomposed in multiple ways. We say that the relation R is finite if the number of such decompositions is finite. Given a set of kernels $K_d : \mathcal{X}_d \times \mathcal{X}_d \to \mathbb{R}$, one for each of the parts of x, the *R-convolution* kernel is defined as

$$K_{R,\otimes}(x, z) = \sum_{(x_1,\ldots,x_D)\in R^{-1}(x)} \sum_{(z_1,\ldots,z_D)\in R^{-1}(z)} \prod_{d=1}^{D} K_d(x_d, z_d) \tag{5}$$

where the sums run over all the possible decompositions of x and z. Similarly, one could use direct sum obtaining

$$K_{R,\oplus}(x, z) = \sum_{(x_1,\ldots,x_D)\in R^{-1}(x)} \sum_{(z_1,\ldots,z_D)\in R^{-1}(z)} \sum_{d=1}^{D} K_d(x_d, z_d). \tag{6}$$

For finite relations R, these functions can be shown to be valid kernels:

Theorem 4 (Haussler 1999) *For any finite R on a space $\mathcal{X}$, the functions $K_{R,\otimes} : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ (defined by Equation (5)) and $K_{R,\oplus} : \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ (defined by Equation (6)) are positive semi-definite kernels on $\mathcal{X} \times \mathcal{X}$.*

Proof: Follows from closure properties of tensor product and direct sum. See Haussler (1999) for details.

The *set kernel* (Shawe-Taylor and Cristianini, 2004) is a special case of convolution kernel that will prove useful in defining kernels between visitor traces. Suppose instances are sets and let us define the part-of relation as the usual set-membership. The kernel over sets K_{set} is then obtained from kernels between set members K_{member} as follows:

$$K_{set}(x, z) = \sum_{\xi \in x} \sum_{\zeta \in z} K_{member}(\xi, \zeta). \tag{7}$$

In order to reduce the dependence on the dimension of the objects, kernels over discrete structures are often normalized. A common choice is that of using normalization in feature space, i.e., given a convolution kernel K_R:

$$K_{norm}(x, z) = \frac{K_R(x, z)}{\sqrt{K_R(x, x)}\sqrt{K_R(z, z)}}. \tag{8}$$

In the case of set kernels, an alternative is that of dividing by the cardinalities of the two sets, thus computing the mean value between pairwise comparisons:[4]

$$K_{mean}(x, z) = \frac{K_{set}(x, z)}{|x||z|}. \tag{9}$$

Richer families of kernels on data structures can be formed by applying composition to the feature mapping induced by a convolution kernel. For example, a convolution kernel K_R can be combined with a Gaussian kernel as follows:

$$K(x, z) = \exp\left(-\gamma\Big(K_R(x, x) - 2K_R(x, z) + K_R(z, z)\Big)\right). \tag{10}$$

4.2 Kernels over Visitor Programs

Going back to the framework defined in Section 3, let $\mathcal{X}$ be a set of first-order logic objects and for $x, z \in \mathcal{X}$ consider the program traces T_x and T_z defined by Equations (3) and (4). In order to define the kernel over program traces we follow a top-down approach. We begin by decomposing traces into parts associated with different visitors (i.e., the elements of the tuple in Equation (4)) and applying a decomposition kernel based on direct sum as defined by Equation (6):

$$K(T_x, T_z) = \sum_{l=1}^{N} K_l(T_{l,x}, T_{l,z}). \tag{11}$$

4. Note that normalizations such as those of Equations (8) and (9) can give indefinite results iff one of the two arguments (say x) is the null vector of the feature space associated to the original kernel (i.e., K_R or K_{set}). In such a case, we will define $K_{norm}(x, z) = K_{mean}(x, z) = 0\ \forall z \in \mathcal{X}, z \neq x$.

Note that there is a unique decomposition of T_x and T_y, that is we just compare proofs of the same visitor. According to Definition 3 for each $l = 1, \ldots, N$, the arguments to K_l are sets of proof trees. Hence, using the set kernel of Equation (7) we further obtain:

$$K_l(T_{l,x}, T_{l,z}) = \sum_{p=1}^{s_{l,x}} \sum_{q=1}^{s_{l,z}} K_{tree}(T_{lp,x}, T_{lq,z}). \tag{12}$$

In this way, we have shown that the problem boils down to defining a kernel K_{tree} over individual proof trees. This will be detailed in the remainder of this section. Note that we can define different kernels for proof trees originating from different visitors.

At the highest level of kernel between visitor programs (Equation (11)), it is advisable to employ a feature space normalization using Equation (8). In some cases it may also be useful to normalize lower-level kernels, in order to rebalance contributions of individual parts. In particular, the mean normalization of Equation (9) can be applied to the kernel over individual visitors (Equation (12)) and it is also possible to normalize kernels between individual proof trees, in order to reduce the influence of the proof size.

4.3 Representing Proof Trees as Prolog Ground Terms

Proof trees are discrete data structures and, in principle, existing kernels on trees could be applied (e.g. Collins and Duffy, 2002; Viswanathan and Smola, 2003). However, we can gain more expressiveness by representing individual proof trees as typed Prolog ground terms. In so doing we can exploit type information on constants and functors so that different sub-kernels can be applied to different object types. In addition, while traditional tree kernels would typically compare *all* pairs of subtrees between two proofs, the kernel on ground terms presented below results in a more selective approach that compares certain parts of two proofs only when reached by following similar inference steps (a distinction that would be difficult to implement with traditional tree kernels).
We will use the following procedure to represent a proof tree as a Prolog ground term:

- Base step: if a node contains a fact, this is already a ground term.
- Induction: if a node contains a clause, then let n be the number of arguments in the head and m the number of atoms in the body (corresponding to the m children of the node). A ground compound term t having $n + 1$ arguments is then formed as follows:
  - the functor name of t is the functor name of the head of the clause;
  - the first n arguments of t are the arguments of the clause head;
  - the last argument of t is a compound term whose functor name is a Prolog constant obtained from the clause number,[5] and whose m arguments are the ground term representations of the m children of the node.

Example 3 *Consider the proof tree of Figure 2 in the mutagenesis domain. The transformation outlined above yields the following representation as a Prolog ground term:*

5. Since numbers cannot be used as functor names, this constant can be simply obtained by prefixing the clause number by '`cbody`'.

```
visit(d26,
      cbody4(cycle(d26,
                   d26_1,
                   cbody1(path(d26,
                               d26_1,
                               d26_6,
                               [d26_1],
                               cbody3(...)),
                          bond(d26,d26_6,d26_1,7))))).
```

where we skipped the representation of the children of path *for the sake of readability.*

We are now able to employ kernels on Prolog ground terms as defined in Passerini and Frasconi (2005) to compute kernels over individual proof trees.

4.4 Kernels on Prolog Ground Terms

We begin with kernels on untyped terms. Let $\mathcal{C}$ be a set of constants and $\mathcal{F}$ a set of functors, and denote by $\mathcal{U}$ the corresponding Herbrand universe (the set of all ground terms that can be formed from constants in $\mathcal{C}$ and functors in $\mathcal{F}$). Let $f^{/n} \in \mathcal{F}$ denote a functor having name f and arity n.

Definition 5 (Sum kernels on untyped terms) *The kernel between two terms t and s is a function $K : \mathcal{U} \times \mathcal{U} \mapsto \mathbb{R}$ defined inductively as follows:*

- *if $s \in \mathcal{C}$ and $t \in \mathcal{C}$ then*
$$K(s,t) = \kappa(s,t) \tag{13}$$
where $\kappa : \mathcal{C} \times \mathcal{C} \mapsto \mathbb{R}$ is a valid kernel on constants;

- *else if s and t are compound terms and have different functors, i.e., $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, then*
$$K(s,t) = \iota(f^{/n}, g^{/m}) \tag{14}$$
where $\iota : \mathcal{F} \times \mathcal{F} \mapsto \mathbb{R}$ is a valid kernel on functors;

- *else if s and t are compound terms and have the same functor, i.e., $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$, then*
$$K(s,t) = \iota(f^{/n}, f^{/n}) + \sum_{i=1}^{n} K(s_i, t_i) \tag{15}$$

- *in all other cases $K(s,t) = 0$.*

Functions κ and ι are called *atomic* kernels as they operate on non-structured symbols. A special but useful case is the atomic delta kernel δ defined as $\delta(x,z) = 1$ if $x = z$ and $\delta(x,z) = 0$ if $x \neq z$.

Example 4 *Consider the two lists* $s = [\mathtt{a}, \mathtt{b}, \mathtt{c}]$ *and* $t = [\mathtt{a}, \mathtt{c}]$*. Recall that in Prolog* $[\mathtt{a}, \mathtt{b}]$ *is a shorthand for* $.(\mathtt{a}, .(\mathtt{b}, []))$ *where the functor* $./2$ *is a data constructor for lists and* $[]$ *is the data constructor for the empty list. Suppose* $\iota(./2, ./2) = 0.25$ *and* $\kappa(x, z) = \delta(x, z)$ *for all* $x, z \in \mathcal{C}$*. Then*

$$\begin{aligned} K(s,t) &= K(.(\mathtt{a}, .(\mathtt{b}, .(\mathtt{c}, []))), .(\mathtt{a}, .(\mathtt{c}, []))) \\ &= \iota(./2, ./2) + K(\mathtt{a}, \mathtt{a}) + K(.(\mathtt{b}, .(\mathtt{c}, [])), .(\mathtt{c}, [])) \\ &= \iota(./2, ./2) + \kappa(\mathtt{a}, \mathtt{a}) + \iota(./2, ./2) + \kappa(\mathtt{b}, \mathtt{c}) + K(.(\mathtt{c}, []), []) \\ &= 0.25 + 1 + 0.25 + 0 + 0 = 1.5 \end{aligned}$$

The result obtained in the above example is similar to what would be achieved with the kernel on higher-order logic basic terms defined in Gärtner et al. (2004). The following examples illustrate the case of two other common data structures.

Example 5 *Consider the two tuples simulated via a predicate* $\mathtt{r}$*:* $s = \mathtt{r}(\mathtt{a}, \mathtt{b}, \mathtt{c})$ *and* $t = \mathtt{r}(\mathtt{d}, \mathtt{b}, \mathtt{a})$*. Suppose* $\iota(\mathtt{r}/3, \mathtt{r}/3) = 0$ *and* $\kappa(x, z) = \delta(x, z)$ *for all* $x, z \in \mathcal{C}$*. Then it immediately follows from the definition that* $K(s, t) = 1$*.*

Example 6 *As a last example consider data structures intended to describe scientific references:*

$$\begin{aligned} r &= \mathtt{article}(\mathtt{"Kernels\ on\ Gnus\ and\ Gnats"}, \mathtt{journal}(\mathtt{ggj}, 2004)) \\ s &= \mathtt{article}(\mathtt{"The\ Logic\ of\ Gnats"}, \mathtt{conference}(\mathtt{icla}, 2004)) \\ t &= \mathtt{article}(\mathtt{"Armadillos\ in\ Hilbert\ space"}, \mathtt{journal}(\mathtt{ijaa}, 2004)) \end{aligned}$$

Using $\kappa(x, z) = \delta(x, z)$ *for all* $x, z \in \mathcal{C}$ *and* $\iota(x, z) = \delta(x, z)$ *for all* $x, z \in \mathcal{F}$*, we obtain* $K(r, s) = 1$*,* $K(r, t) = 3$*, and* $K(s, t) = 1$*. The fact that all papers are published in the same year does not contribute to* $K(r, s)$ *or* $K(s, t)$ *since these pairs have different functors describing the venue of the publication; it does contribute to* $K(r, t)$ *as they are both journal papers. Note that strings have been treated as constants (as standard in Prolog). Under our above definition the kernel cannot recognize the fact that* r *and* s *share a word in the title.*

A finer level of granularity in the definition of ground term kernels can be gained from the use of typed terms. This extra flexibility may be necessary to specify different kernel functions associated with constants of different type (e.g., numerical vs categorical). Types are also useful to specify different kernels associated to different arguments of compound terms. As detailed below, this allows us to distinguish different roles played by clauses in a proof tree.

Our approach for introducing types is similar to that proposed by Lakshman and Reddy (1991). We denote by $\mathcal{T}$ the ranked set of type constructors, which contains at least the nullary constructor $\perp$. The type signature of a function of arity n has the form $\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'$ where $n \geq 0$ is the number of arguments, $\tau_1, \ldots, \tau_k \in \mathcal{T}$ are their types, and $\tau' \in \mathcal{T}$ is the type of the result. Functions of arity 0 have signature $\perp \mapsto \tau'$ and can therefore be interpreted as constants of type τ'. The type of a function is the type of its result. The type signature of a predicate of arity n has the form $\tau_1 \times, \ldots, \times \tau_n \mapsto \Omega$ where

$\Omega \in \mathcal{T}$ is the type of Booleans, and is thus a special case of type signatures of functions. We write $t : \tau$ to assert that t is a term of type τ. We denote by $\mathcal{G}$ the set of all typed ground terms, by $\mathcal{C} \subset \mathcal{G}$ the set of all typed constants, and by $\mathcal{F}$ the set of typed functors. Finally we introduce a (possibly empty) set of *distinguished* type signatures $\mathcal{D} \subset \mathcal{T}$ that can be useful to specify ad-hoc kernel functions on certain compound terms.

Definition 6 (Sum kernels on typed terms) *The kernel between two typed terms t and s is defined inductively as follows:*

- *if $s \in \mathcal{C}$, $t \in \mathcal{C}$, $s : \tau$, $t : \tau$ then*

$$K(s,t) = \kappa_\tau(s,t) \tag{16}$$

  where $\kappa_\tau : \mathcal{C} \times \mathcal{C} \mapsto \mathbb{R}$ is a valid kernel on constants of type τ;

- *else if s and t are compound terms that have the same type but different functors or signatures, i.e., $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$, $s : \sigma_1 \times, \ldots, \times \sigma_n \mapsto \tau'$, $t : \tau_1 \times, \ldots, \times \tau_m \mapsto \tau'$, then*

$$K(s,t) = \iota_{\tau'}(f^{/n}, g^{/m}) \tag{17}$$

  where $\iota_{\tau'} : \mathcal{F} \times \mathcal{F} \mapsto \mathbb{R}$ is a valid kernel on functors that construct terms of type τ'

- *else if s and t are compound terms and have the same functor and type signature, i.e., $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and $s, t : \tau_1 \times, \ldots, \times \tau_n \mapsto \tau'$, then*

$$K(s,t) = \begin{cases} \kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'}(s,t) \\ \qquad \text{if } (\tau_1 \times, \ldots, \times \tau_n \mapsto \tau') \in \mathcal{D} \\ \iota_{\tau'}(f^{/n}, f^{/n}) + \sum_{i=1}^{n} K(s_i, t_i) \quad \text{otherwise} \end{cases} \tag{18}$$

  where $\kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'} : \mathcal{U} \times \mathcal{U} \mapsto \mathbb{R}$ is a valid kernel on terms having distinguished type signature $\tau_1 \times, \ldots, \times \tau_n \mapsto \tau' \in \mathcal{D}$.

- *in all other cases $K(s,t) = 0$.*

Versions of the kernels which combine arguments using products instead of sums can be easily defined as follows.

Definition 7 (Product kernels on untyped terms) *Use Definition 5 replacing Equation (15) with*

$$K(s,t) = \iota(f^{/n}, f^{/n}) \prod_{i=1}^{n} K(s_i, t_i) \tag{19}$$

Definition 8 (Product kernels on typed terms) *Use Definition 6 replacing Equation (18) with*

$$K(s,t) = \begin{cases} \kappa_{\tau_1 \times, \ldots, \times \tau_n \mapsto \tau'}(s,t) \\ \qquad \text{if } (\tau_1 \times, \ldots, \times \tau_n \mapsto \tau') \in \mathcal{D} \\ \iota_{\tau'}(f^{/n}, f^{/n}) \prod_{i=1}^{n} K(s_i, t_i) \quad \text{otherwise} \end{cases} \tag{20}$$

The families of functions in Definitions 5–8 are special cases of Haussler's decomposition kernels and therefore they are positive semi-definite (see Appendix A for formal results).

4.5 Kernels on Prolog Proof Trees

In order to employ full typed term kernels (as in Definitions 6 and 8) on proof trees, we need a typed syntax for their ground term representation. We will assume the following default types for constants: `num` (numerical) and `cat` (categorical). Types for compounds terms will be either `fact`, corresponding to leaves in the proof tree, `clause` in the case of internal nodes, and `body` when containing the body of a clause. Note that regardless of the specific implementation of kernels between types, such definitions imply that we actually compare the common subpart of proofs starting from the goal (the visitor clause), and stop whenever the two proofs diverge.

A number of special cases of kernels can be implemented with appropriate choices of the kernel for compound and atomic terms. The *equivalence* kernel outputs one iff two proofs are equivalent, and zero otherwise:

$$K_{equiv}(s,t) = \begin{cases} 1 & \text{if } s \equiv t \\ 0 & \text{otherwise} \end{cases} \tag{21}$$

We say that two proofs are equivalent if the same sequence of clauses is proven in the two cases, and the head arguments in corresponding clauses satisfy a given equivalence relation. A trivial implementation of proof equivalence can be obtained using the product kernel on typed terms (Definition 8) in combination with the delta kernel on constants and functors.

In many cases, we will be interested in ignoring some of the arguments of a pair of ground terms when computing the kernel between them. As an example, consider the atom bond representation of a molecule shown in the upper part of Figure 1. The first arguments of `atm` and `bond` predicates are simply molecule and atom identifiers, and we would like to ignore their values when comparing two molecules together. This can be implemented using a special `ignore` type for arguments that should be ignored in comparisons, and a corresponding *constant* kernel which always outputs a constant value:

$$K_\eta(s,t) = \eta.$$

It is straightforward to see that K_η is a valid kernel provided $\eta \geq 0$. The constant η should be set equal to the identity element of the operation used to combine results for the different arguments of the term under consideration, that is $\eta = 0$ for the sum kernel and $\eta = 1$ for the product one.

The extreme use for this kernel is that of implementing the notion of *functor equality* for proof tree nodes, where two nodes are the same iff they share the same functor, regardless of the specific values taken by their arguments. Given two ground terms $s = f(s_1, \ldots, s_n)$ and $t = g(t_1, \ldots, t_m)$ the functor equality kernel is given by:

$$K_f(s,t) = \begin{cases} 0 & \text{if } type(s) \neq type(t) \\ \delta(f^{/n}, g^{/m}) & \text{if } s,t : \texttt{fact} \\ \delta(f^{/n}, g^{/m}) \star K(s_n, t_m) & \text{if } s,t : \texttt{clause} \\ K(s,t) & \text{if } s,t : \texttt{body} \end{cases} \tag{22}$$

where K is a kernel on ground terms as defined in Section 4.4, and the operator $\star$ can be either sum or product. Note that if s and t represent clauses (i.e., internal nodes of the

proof tree), the comparison skips clause head arguments, represented by the first $n-1$ (resp. $m-1$) arguments of the terms, and compares the bodies (the last argument, see Section 4.3) thus proceeding on the children of the nodes. This kernel allows to define a non trivial equivalence between proofs (or parts of them) checking which clauses are proved in sequence and ignoring the specific values of their head arguments.

Moreover, it will often be useful to define custom kernels for specific terms by using distinguished type signatures. Appendix B contains details of possible kernel configurations as sets of Prolog clauses, while Appendix C contains the Prolog code for all visitors and kernel configurations employed in the experimental section.

5. Experiments

We run a number of experiments in order to demonstrate the possibilities of the proposed method. In particular, we aim to empirically show that

1. statistical learning in the ILP setting can be addressed, scaling better than typical ILP algorithms with the complexity of the target hypothesis;
2. problems which are difficult for traditional ILP algorithms can be solved;
3. both classification and regression tasks can be effectively handled;
4. significant improvements on real world applications can be achieved.

For classification tasks, we employed SVM (Cortes and Vapnik, 1995) using the Gist[6] implementation, which permits to separate kernel calculation from training by accepting the complete kernel matrix as input. We compared our method with two popular and diverse ILP algorithms: Tilde (Blockeel and De Raedt, 1998), which upgrades C4.5 to induction of logical decision trees, and Progol (Muggleton, 1995), which learns logical theories using inverse entailment.

Regression is quite a difficult task for ILP techniques, and few algorithms currently exist which are able to address it. Conversely, our definition of kernel over proof trees allows us to apply standard kernel methods for regression, such as kernel ridge regression (KRR, (Poggio and Smale, 2003)) and support vector regression (Vapnik, 1995). We report results using the former approach, as training was more stable and no significant difference in performance could be noted. However, when dealing with large data sets, the latter method would be preferable for efficiency reasons. In Section 5.4 we report regression experiments comparing our approach to a number of propositional as well as relational learners.

5.1 Bongard Problems

In order to provide a full basic example of visitor program construction and exploitation of the proof tree information, we created a very simple Bongard problem (Bongard, 1970). The concept to be learned can be represented with the simple pattern *triangle-X^n-triangle* for a given n, meaning that a positive example is a scene containing two triangles nested into

6. The Gist package by W. Stafford Noble and P. Pavlidis is available from `http://microarray.genomecenter.columbia.edu/gist/`.

one another with exactly n objects (possibly triangles) in between. Figure 3 shows a pair of examples of such scenes with their representation as Prolog facts and their classification according to the pattern for $n = 1$.

A possible example of background knowledge introduces the concepts of *nesting* in containment and *polygon* as a generic object, and can be represented as follows:

```
inside(X,A,B):- in(X,A,B).                    % clause nr 1
inside(X,A,B):-                               % clause nr 2
    in(X,A,C),
    inside(X,C,B).
polygon(X,A) :-  triangle(X,A).               % clause nr 3
polygon(X,A) :-  rectangle(X,A).              % clause nr 4
polygon(X,A) :-  circle(X,A).                 % clause nr 5
```

A visitor exploiting such background knowledge, and having hints on the target concept, could be looking for two polygons contained one into the other. This can be represented as:

```
visit(X):-                                    % clause nr 6
    inside(X,A,B),polygon(X,A),polygon(X,B).
```

Figure 4 shows the proofs trees obtained running such a visitor on the first Bongard problem in Figure 3.

A very simple kernel can be employed to solve such a task, namely an equivalence kernel with functor equality for nodewise comparison. For any value of n, such a kernel maps the examples into a feature space where there is a single feature discriminating between positive and negative examples, while the simple use of ground facts without intensional background knowledge would not provide sufficient information for the task.

The data set was generated by creating m scenes each containing a series of ℓ randomly chosen objects nested one into the other, and repeating the procedure for ℓ varying from 2 to 20. Moreover, we generated two different data sets by choosing $m = 10$ and $m = 50$ respectively. Finally, for each data set we obtained 15 experimental settings denoted by $n \in [0, 14]$. In each setting, positive examples were scenes containing the pattern *triangle-*X^n*-triangle*. We run an SVM with the above mentioned proof tree kernel and a fixed value $C = 10$ for the regularization parameter, on the basis that the data set is noise free. We evaluated its performance with a leave-one-out (LOO) procedure, and compared it to the empirical error of Tilde and Progol trained on the same data and background knowledge (including the visitor). Here we focus on showing that ILP algorithms have troubles finding a consistent hypothesis for this problem, hence we did not measure their generalization.

Figure 5(a) plots results for $m = 10$. Both Tilde and Progol stopped learning the concept for $n > 4$. Progol found the trivial empty hypothesis for all $n > 4$ apart from $n = 6$, and Tilde for all $n > 9$. While never learning the concept with 100% generalization accuracy, the SVM performance was much more stable when increasing the nesting level corresponding to positive examples. Figure 5(b) plots results for $m = 50$. Progol was extremely expensive to train with respect to the other methods. It successfully learned the concept for $n \leq 2$, but we stopped training for $n = 3$ after more than one week training time on a 3.20 GHz PENTIUM IV. Tilde stopped learning the concept for $n > 8$, and found the trivial empty hypothesis for $n > 12$. Conversely, the SVM was almost always able to learn the concept with 100% generalization accuracy, regardless of its complexity level.

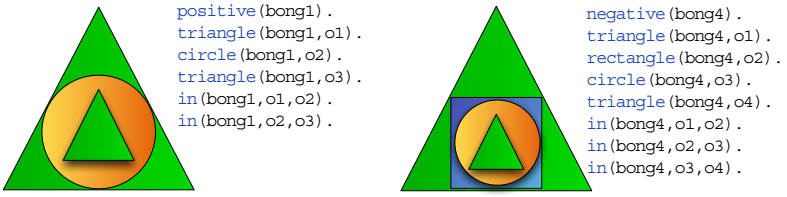

Figure 3: Graphical and Prolog facts representation of two Bongard scenes. The left and right examples are positive and negative, respectively, according to the pattern *triangle-X-triangle*.

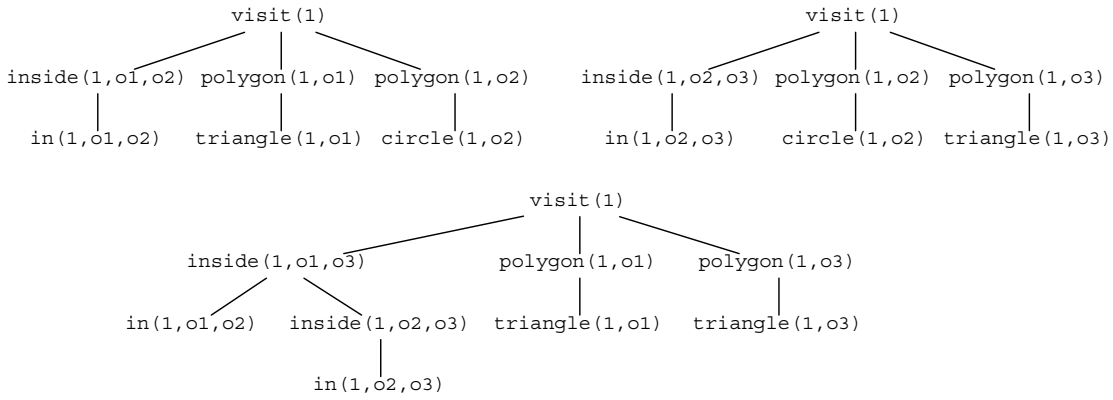

Figure 4: Proof trees obtained by running the visitor on the first Bongard problem in Fig. 3.

Note that in order for the ILP algorithms to learn the target concept regardless of the nesting level, it would be necessary to provide a more informed `inside` predicate, which explicitly contains such nesting level as one of its arguments. The ability of the kernel to extract information from the predicate proof, on the other hand, allows our method to be employed when only partial background knowledge is available, which is typically the case in real world applications.

5.2 M-of-N Problems

The possibility to plug background knowledge into the kernel allows addressing problems that are notoriously hard for ILP approaches. An example of such concepts is the M-of-N one, which expects the model to be able to count and make the decision accordingly.

We represented this kind of tasks with a toy problem. Examples are strings of integers $i \in [0, 9]$, and a string is positive iff more than a half of its pairs of consecutive elements is ordered, where we employ the partial ordering relation $\leq$ between numbers. In this task, M and N are example dependent, while their ratio is fixed.

As background knowledge, we introduced the concepts of "length two substring" and "pairwise ordering":

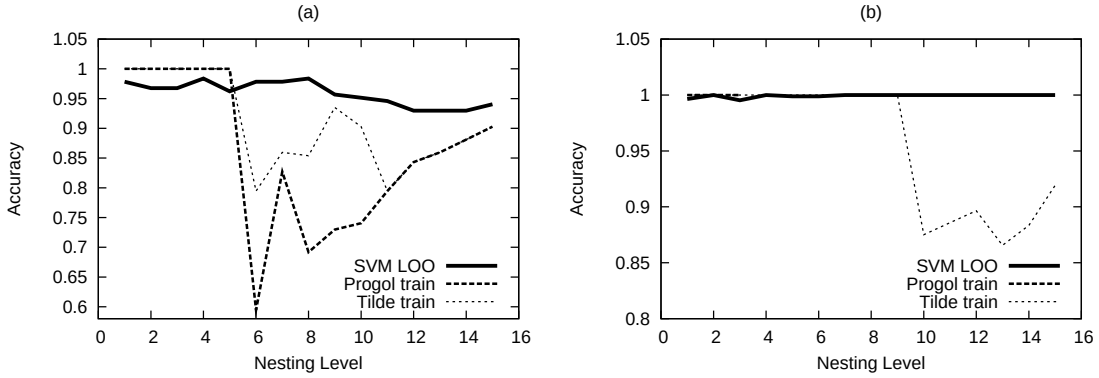

Figure 5: Comparison between SVM leave-one-out error, Progol and Tilde empirical error in learning the *triangle-X^n-triangle* for different values of n, for data sets corresponding to $m = 10$ (a) and $m = 50$ (b).

```
substr([A,B],[A,B|_T]).
substr([A,B],[_H|T]):-
    substr([A,B],T).

comp(A,B):- A @> B.
comp(A,B):- A @=< B.
```

We then designed a visitor which looks for a substring of length two in the example, and compares its elements:

```
visit(X):-
    string(X,S),substr([A,B],S),comp(A,B).
```

We also declared `substr` to be a leaf predicate, thus pruning the proof tree as explained in Section 3.4, because we are not interested in where the substring is located within the example.

The kernel we employed for this task is a sum kernel with functor equality for nodewise comparison. This kernel basically counts the number of clauses proved in the common subpart of two proof trees, where common means that the same clauses were proved regardless of the specific values of their head arguments.

The data set was created in the following way: the training set was made of 150 randomly generated strings of length 4 and 150 strings of length 5; the test set was made of 1455 randomly generated strings of length from 6 to 100. This allowed to verify the generalization performance of the algorithm for lengths very different from the ones it was trained on.

Accuracy on the test set for a default value of the regularization parameter $C = 1$ was 93.5%, with a contingency table as in Table 1. Moreover, false negatives were the nearest to the decision threshold, and slightly modifying the regularization parameter led to 100% accuracy. On the other hand, neither Tilde nor Progol were able to induce any approximation of the target concept with the available background knowledge. A number of problems prevented them from learning:

		-1	1	
	-1	528	0	True
	1	94	833	
		Predicted		

Table 1: Contingency table for the strings task with default regularization parameter. Predicted class is on columns, true class on rows.

1. All proofs of a given predicate (`substr`) were necessary ingredients for the target concept.

2. Counting such proofs was needed, conditioned on the proof details.

3. Gain measures were useless in guiding Tilde hypothesis search, as single atoms forming the target concept had no discriminative power if taken alone.

These problems are due to the need for an aggregation predicate (in this case count) to correctly define the target concept. Dealing with aggregation is known to be difficult for relational learning (Perlich and Provost, 2003; Knobbe et al., 2002).

In order for Progol to learn the target concept, two explicit conditioned counting predicates had to be provided, counting the number of ordered (resp. unordered) length two substrings of a given string. Tilde was still unable to learn the concept with such background knowledge, due the above mentioned problem with gain at intermediate steps of the search, and full lookahead of all building predicates was needed. Again, this is a known problem for decision tree learners (Van de Velde, 1989).

5.3 Protein Fold Classification

In this experiment, we tested our methodology on the protein fold classification problem studied by Turcotte et al. (2001). The task consists of classifying proteins into Scop folds, given their high-level logical descriptions about secondary structure and amino acid sequence. Scop is a manually curated database of proteins hierarchically organized according to their structural properties. At the top level Scop groups proteins into four main classes (all-α, all-β, α/β, and $\alpha+\beta$). Each class is then divided into folds that group together proteins with similar secondary structures and three-dimensional arrangements. We used the data set made available as a supplement to the paper by Turcotte et al. (2001)[7] that consists of the five most populated folds from each of the four main Scop classes. This setting yields 20 binary classification problems. The data sets for each of the 20 problems are relatively small (from about 30 to about 160 examples per fold, totaling 1143 examples).

We relied on the background knowledge provided in Turcotte et al. (2001), to design a set of visitors managing increasingly complex information. A global visitor was used to extract protein level information, such as its length and the number of its α or β secondary structure segments. A local visitor explored the details of each of such segments, while a connection visitor looked for pairs of adjacent segments within the protein. Numerical

7. `http://www.bmm.icnet.uk/ilp/data/ml_2000.tar.gz`.

values were normalized within each top level fold class. The kernel configuration mainly consisted of type signatures aiming to ignore identifiers and treat some of the numerical features as categorical ones. A functor equality kernel was employed for those nodes of the proofs which did not contain valuable information in their arguments. Code details for visitors and kernel configuration can be found in Appendix-C.3.

Following Turcotte et al. (2001), we measured prediction accuracy by 10-fold cross-validation, micro-averaging the results over the 20 experiments by summing contingency tables. The proof-tree kernel was combined with a Gaussian kernel (see Equation (10)) in order to model nonlinear interactions between the features extracted by the visitor program. Model selection (i.e., choice of the Gaussian width γ and the SVM regularization parameter C) was performed for each binary problem with a LOO procedure before running the 10-fold cross validation. Table 2 shows comparisons between the best setting for Progol (as reported by Turcotte et al. (2001)), which uses both propositional and relational background knowledge, results for Tilde using the same setting, and SVM with our kernel over proof trees. The difference between Tilde and Progol is not significant, while our SVM achieves significantly higher overall accuracy with respect to both methods.

5.4 QSAR Regression Tasks

Quantitative structure activity relationship (QSAR) tasks deal with the problem of predicting the biological activity of a molecule given its chemical structure. They can thus be naturally represented as regression problems. The chemical structure of molecules is typically represented by atom and bond predicates, possibly specifying also non topological attributes such as atom and bond detailed types and atom partial charge. Additional features include molecule physico-chemical properties, such as its weight, its hydrophobicity ($logP$) and $lumo$, which is the energy of the molecule lowest unoccupied orbital. Intensional background knowledge can be represented by predicates looking for ring structures and functional groups within the molecule, such as benzene, anthracene and nitro. Relational features can also be propositionalized in different ways in order to employ propositional learners.

In the following we focused on two well known QSAR data sets, mutagenesis and biodegradability, and compared to published results for different relational and propositional learners, always attaining to their same experimental settings. In both cases we run a preliminary model selection phase (optimizing Gaussian width and regularization parameter) on an additional 10 fold cross validation procedure. We employed the Pearson correlation coefficient as a standard performance measure, and two tailed Fisher z tests at 0.05 significance level in order to verify if the performance difference between pairs of methods was statistically significant.

5.4.1 Mutagenesis

The mutagenicity problem is a standard benchmark for ILP approaches. The problem is treated in Srinivasan et al. (1996) as a binary classification task (mutagenic vs. non-mutagenic). Here we focused on its original formulation as a regression task, and compared to the results presented in Kramer (1999) for the regression friendly data set.

	Tilde	Progol	SVM
All-α:			
Globin-like	97.4	95.1	94.9
DNA-binding 3-helical bundle	81.1	83.0	88.9
4-helical cytokines	83.3	70.7	86.7
lambda repressor-like DNA-binding domains	70.0	73.4	83.3
EF Hand-like	71.4	77.6	85.7
All-β:			
Immunoglobulin-like beta-sandwich	74.1	76.3	85.2
SH3-like barrel	91.7	91.4	93.8
OB-fold	65.0	78.4	83.3
Trypsin-like serine proteases	95.2	93.1	93.7
Lipocalins	83.3	88.3	92.9
α/β:			
beta/alpha (TIM)-barrel	69.7	70.7	73.3
NAD(P)-binding Rossmann-fold domains	79.4	71.6	84.1
P-loop containing nucleotide triphosphate hydrolases	64.3	76.0	76.2
alpha/beta-Hydrolases	58.3	72.2	86.1
Periplasmic binding protein-like II	79.5	68.9	79.5
$\alpha+\beta$:			
Interleukin 8-like chemokines	92.6	92.9	96.3
beta-Grasp	52.8	71.7	88.9
Ferredoxin-like	69.2	83.1	76.9
Zincin-like	51.3	64.3	79.5
SH2-like	82.1	76.8	66.7
Micro average:	75.2 $\pm$2.5	78.3 $\pm$2.4	83.6 $\pm$2.2

Table 2: Protein fold classification: 10-fold cross validation accuracy (%) for Tilde, Progol and SVM for the different classification tasks, and micro averaged accuracies with 95% confidence intervals. Results for Progol are taken from Turcotte et al. (2001).

System	r
KRR	**0.898**(**0.002**)
S-CART	0.830 (0.020)
P + S-CART	0.834 (0.010)
P + M5'	**0.893**(**0.001**)
P + SP + S-CART	0.767 (0.038)
P + SP + M5'	0.835 (0.012)

Table 3: Pearson correlation coefficient for the different learners on the regression friendly mutagenesis data set. Results are averaged over four 10-fold cross validation procedures, and standard deviations over the four procedures are reported. Boldface numbers are significantly better than plain ones. All other differences are not significant. Results for all systems except for KRR are taken from Kramer (1999).

We employed a global visitor exploring physico-chemical properties of the molecule, that is *logp*, *lumo*, *ind1* and *inda*. We then developed a set of visitors exploiting the ring theory for nitro aromatic and heteroaromatic compounds, each looking for compounds of a certain type, and extracting the properties of the atoms belonging to it. We employed pruned trees for such visitors, as described in the example shown in Section 3.4. Kernel configuration was mostly made of type signatures as for the protein fold classification task (Section 5.3, see Appendix-C.4 for code details).
Competing algorithms included S-CART (Kramer, 1999), which is an upgrade of CART to first order logic, and M5' (Quinlan, 1993; Wang and Witten, 1997), a propositional regression-tree induction algorithm. Propositionalization was conducted either by (P) counting occurrences of different functional groups (together to physico-chemical global properties), or (SP) running a supervised stochastic propositionalization algorithm as described in Kramer (1999). Table 3 reports experimental comparisons on four 10 fold cross validation procedures. Our method consistently outperforms all other learners, and such difference is significant on four out of five cases.

5.4.2 Biodegradability

Degradation is the process by which chemicals are transformed into components which are not considered pollutants. A number of different pathways are responsible for such process, depending on environmental conditions. Blockeel et al. (2004) conducted a study focused on aqueous biodegradation under aerobic conditions. Low and high estimates of half life time degradation rate were collected for 328 molecules. The regression task consisted in predicting the natural logarithm of the arithmetic mean of the low and high estimate for a given molecule. A comprehensive background knowledge of rings and functional groups was available as for the mutagenesis data set. Moreover, relational features had been propositionalized in two different ways. Four sets of features were thus made available to learning algorithms (Blockeel et al., 2004):

- *Global* consisted of molecule physico-chemical properties, namely weight and logP.

System	G	G+P1	G+P2	G+R	G+P1+P2+R
KRR	0.472 (0.005)	0.701 (0.005)	0.683 (0.006)	0.694 (0.005)	0.695 (0.005)
Tilde	0.487 (0.020)	0.596 (0.029)	0.615 (0.014)	0.616 (0.021)	0.595 (0.020)
S-CART	0.476 (0.031)	0.563 (0.010)	0.595 (0.032)	0.605 (0.023)	0.606 (0.032)
M5'	0.503 (0.012)	0.579 (0.024)	0.646 (0.013)		
LR	0.436 (0.004)	0.592 (0.014)	0.443 (0.026)		

Table 4: Pearson correlation coefficient for the different learners for various combinations of features on the biodegradability data set. Results are averaged over five 10-fold cross validation procedures, and standard deviations over the five procedures are reported. Results for all systems except for KRR are taken from Blockeel et al. (2004).

- *P1* were counts of rings and functional groups defined in the background theory.
- *P2* were counts of small substructures of molecules (all connected substructures of two or three atoms, those of four with a star topology).
- R contained full relational features: atoms, bonds, ring and functional structures described by their constituent atoms and those connecting them to the rest of the molecule.

We developed appropriate visitors for each of these feature sets. Visitors for full relational features (R) explored atoms within rings and functional structures as in the mutagenesis task, additionally including information about atoms connecting each compound to the rest of the molecule. Numerical features[8] were normalized. The kernel configuration was again similar to that in the protein fold classification task (Section 5.3), but we also modified the default combining operator for a few type signatures in order to compared substructures of the same type only (code details in Appendix-C.5).

A number of relational and propositional learners were compared in Blockeel et al. (2004) on different feature sets: apart from S-CART and M5', already introduced for the mutagenesis data set, simple linear regression (LR) and the version of Tilde learning regression trees (Blockeel and De Raedt, 1998). Table 4 reports average and standard deviation of Pearson correlation coefficient on five 10-fold cross validation procedures, for different combinations of the feature sets. Our kernel outperforms all other methods on four out of five scenarios, and in two cases results are significantly better than any competitor (see Figure 6).

In a second batch of experiments, Blockeel et al. (2004) separately predicted low and high estimates of half life time degradation rate, and reported the mean of such predictions. Results are shown in Table 5. While other methods often improve their performance over the previous batch, our method is almost unaffected. Still, it outperforms all learners on the same four scenarios, and in one case it obtains significantly better results than any other algorithm (Figure 7).

8. Apart from those in *P1* which had a small range ($[0, 4]$).

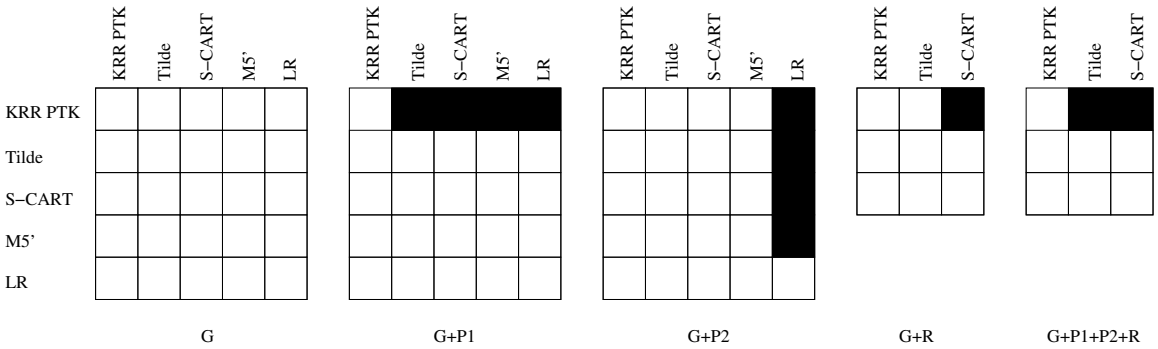

Figure 6: Significance of performance difference between learners for the biodegradability data set. A black box indicates that the learner on the row is significantly better than that on the column for the given feature setting.

System	G	G+P1	G+P2	G+R	G+P1+P2+R
KRR	0.498 (0.004)	0.700 (0.005)	0.683 (0.006)	0.694 (0.005)	0.695 (0.005)
Tilde	0.495 (0.015)	0.612 (0.022)	0.619 (0.021)	0.635 (0.018)	0.618 (0.022)
S-CART	0.478 (0.016)	0.581 (0.015)	0.636 (0.015)	0.659 (0.019)	0.631 (0.026)
M5'	0.502 (0.014)	0.592 (0.013)	0.646 (0.014)		
LR	0.437 (0.005)	0.592 (0.013)	0.455 (0.022)		

Table 5: Pearson correlation coefficient for the different learners for various combinations of features on the biodegradability data set (second batch). Results are averaged over five 10-fold cross validation procedures, and standard deviations over the five procedures are reported. Results for all systems except for KRR are taken from Blockeel et al. (2004).

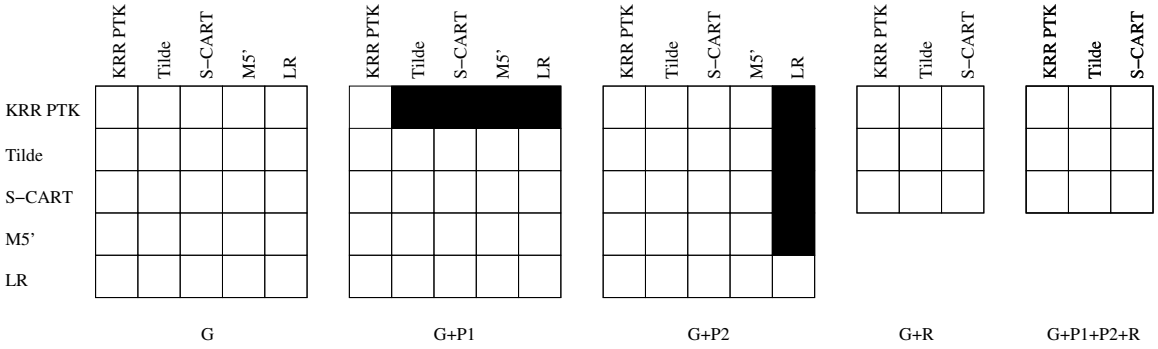

Figure 7: Significance of performance difference between learners for the biodegradability data set (second batch). A black box indicates that the learner on the row is significantly better than that on the column for the given feature setting.

6. Discussion and Related Work

When tackling inductive learning problems using the presented techniques, there are a number of design decisions to be made. These include: the choice of the background theory $\mathcal{B}$, visitor program $\mathcal{V}$ and also the kernel K. As compared to traditional ILP, the background theory $\mathcal{B}$ is similar, the visitor program plays the role of the declarative and inductive bias, and the kernel can perhaps be related to some distance based learning approaches (Ramon and Bruynooghe, 1998; Horvath et al., 2001). The visitor program, however, constitutes an entirely different form of bias than the typical declarative language bias employed in ILP (Nédellec et al., 1996), which is purely syntactic. The visitor incorporates a much more procedural bias, which is perhaps more similar to explanation-based learning (Mitchell et al., 1986). Indeed, explanation-based learning also starts from proof trees or traces for specific examples and then generalizes them using deductive methods (such as regression or partial evaluation (Van Harmelen and Bundy, 1988)). There has, however, also been a strong interest in integrating inductive concept-learning with explanation-based learning (Mitchell et al., 1986). To some extent, the combination of the proof trees with the kernel realizes such an integration, although it does not produce interpretable rules, due to the use of the kernel. The notion and use of background theory has—to some degree—always been debated. The reason is that, on the one hand, it provides the user with a flexible means to guide and influence the learning process, but, on the other hand, it is not always easy to define a background theory that will yield accurate hypotheses. For instance, in applying traditional ILP systems to the Bongard example (Section 5.1), it is clear that by using only the outcome of the `inside` predicate, one loses the information of how many objects are between the outermost and innermost triangle. But this could easily be fixed by defining `inside(X,Y,Z)` as "X is inside Y with Z objects between them." In other words, a change of background definitions makes it possible to learn the correct concept, even by traditional ILP systems. This is one example that shows that background theory is a powerful but sometimes hard to master tool.

To gain some further insights into the relationship of our method to traditional ILP, let us try to relate the background theory to the visitor program. From an ILP perspective, it does seem natural to add the visitor program to the background theory and run the traditional ILP system. Whereas this is—in principle—possible, there are some major differences that would result. Indeed, the ILP system could only use the predicates mentioned in the visitor program as conditions in its hypotheses, and it would not have any means to look into the trace or proof. For instance, if there is a predicate v in the visitor program that is defined using two different clauses, the ILP system will not be able to distinguish instances of v proven using the first clause from those proven using the second one. Also, differences and similarities between proof trees could not be discovered unless one would also add the meta-program (implemented as a Prolog predicate) that generates the proof tree to the background theory. In this case, the information about the structure of proof trees and the clauses being used in there could be employed in the hypotheses of the ILP system. This would yield conditions such as `prove(visitor(x), proof-tree)`. However, since ILP systems have severe search problems when dealing with large structured and terms and recursion, this idea cannot be applied in practice. The use of kernels to realize the generalization is much more appealing because there is no search involved in computing the

kernel. Finally, let us remark that it would be interesting to further investigate the design choices $(\mathcal{B}, \mathcal{V}, K)$ to be made. In particular, one may wonder under what conditions two possible choices (say $(\mathcal{B}, \mathcal{V}, K)$ and $(\mathcal{B}', \mathcal{V}', K')$) are equivalent, and whether this would allow us to reformulate one element (say the visitor) as a part of another one (say the background theory).

7. Conclusions

We have introduced the general idea of kernels over program traces and specialized it to the case of Prolog proof trees in the logic programming paradigm. The theory and the experimental results that we have obtained indicate that this method can be seen as a successful attempt to bridge several aspects of symbolic and statistical learning, including the ability of working with relational data, the incorporation of background knowledge in a flexible and principled way, and the use of regularization. Computational complexity is also an advantage compared to typical ILP systems. The kernel matrix can be computed in time quadratic in the size of the training set and the complexity of the learning problem is that of the kernel method employed (e.g., SVM or KRR) which is typically inferior to ILP algorithms. This may potentially open the road towards some large-scale applications of learning in the ILP setting.

The advantages of the proposed approach were experimentally verified. The Bongard problems showed that our method scales better than typical ILP algorithms with the complexity of the target concept. Furthermore, it is able to effectively address problems (like the M-of-N one) that require precise counting, and are difficult to solve with classic ILP approaches. Both classification and regression tasks can be naturally handled using appropriate kernel methods. Finally, the robust nature of statistical learning can offer advantages with respect to symbolic approaches when dealing with noisy data sets, as shown by the improved performance on the bioinformatics and chemoinformatics tasks.

Besides the cases of classification and regression that have been studied in this paper, other learning tasks could naturally benefit from the proposed framework including clustering, ranking, and novelty detection. One advantage of ILP as compared to the present work is the intrinsic ability of ILP to *generate* transparent explanations of the learned function. Developing kernel machines capable of providing transparent predictions and the use of kernel-based approaches to guide hypothesis search as in ILP remain interesting open issues.

Acknowledgments

This research is supported by EU Grant APrIL II (contract n° 508861). PF and AP are also partially supported by MIUR Grant 2003091149_002. We would like to thank the anonymous reviewers whose comments contributed to improve the paper substantially.

Appendix A. Proofs of Theorems

We give in this appendix a result showing that the class of functions studied in this paper are positive semi-definite and therefore valid Mercer kernels.

Theorem 9 *The kernel function on Prolog ground terms given in Definition 5 is positive semi-definite.*

Proof. Let us introduce the following decomposition structure (see Shawe-Taylor and Cristianini, 2004): $\mathcal{R} = \langle (X_1, X_2), R, (k_1, k_2) \rangle$ with $X_1 = \mathcal{F}$, $X_2 = (\mathcal{F}, \mathcal{U})$, and

$$R = \left\{ (f^{/n}, (f^{/n}, a), s) \text{s.t. } s \text{ is a term having functor } f^{/n} \text{ and tuple of arguments } a \right\}.$$

Then it can be immediately verified that the kernel function of Equations (14) and (15) correspond to the direct sum decomposition kernel associated with the decomposition structure $\mathcal{R}$ if $k_1 = \iota$ and $k_2((f^{/n}, a), (g^{/m}, b)) = \delta(f^{/n}, g^{/m}) k'(a, b)$ where given $a = (s_1, \ldots, s_n)$ and $b = (t_1, \ldots, t_n)$

$$k'(a, b) = \sum_{i=1}^{n} K(s_i, t_i).$$

Note that k' is a valid kernel if K is (being a direct sum). The proof then follows by induction using the fact that kernels for base steps (κ (Equation (13)) and ι (Equation (14))) are by hypothesis positive semi-definite, and the induction step simply consists of combining positive semi-definite kernels by direct sum which itself produces valid kernels (Theorem 4). □

Theorem 10 *The kernel function on typed Prolog ground terms given in Definitions 6 is positive semi-definite.*

Proof. We can use the same technique as for Theorem 9 but including types in the decomposition structure: $\mathcal{R} = \langle (X_1, X_2), R, (k_1, k_2) \rangle$ with $X_1 = (\mathcal{F}, \mathcal{T})$, $X_2 = (\mathcal{F}, \mathcal{T}, \mathcal{U})$, and

$$R = \{((f^{/n}, \tau), (f^{/n}, (\tau_1 \times, \ldots, \times \tau_n \mapsto \tau), t), s) \text{ s.t. } s \text{ is a term having functor } f^{/n}, \\ \text{tuple of arguments } t, \text{ and type signature } \tau_1 \times, \ldots, \times \tau_n \mapsto \tau\}.$$

The kernel function of Equations (17) and (18) correspond to the direct sum decomposition kernel associated with the decomposition structure $\mathcal{R}$ if:

$$k_1((f^{/n}, \tau), (g^{/m}, \sigma)) = \delta(\tau, \sigma) \iota_\tau(f^{/n}, g^{/m})$$

and

$$k_2((f^{/n}, \tau_1 \times, \ldots, \times \tau_n \mapsto \tau, a), (g^{/m}, \sigma_1 \times, \ldots, \times \sigma_m \mapsto \sigma, b)) = \\ \delta(f^{/n}, g^{/m}) \delta(\tau_1 \times, \ldots, \times \tau_n \mapsto \tau, \sigma_1 \times, \ldots, \times \sigma_m \mapsto \sigma) k'(a, b).$$

The proof follows from Theorem 4 and by induction using the fact that κ_τ (Equation (16)), ι_τ (Equation (17)) and kernels on distinguished types (see Equation (18)) are by hypothesis valid kernels. □

Theorem 11 *The kernel functions on Prolog ground terms given in Definitions 7 and 8 are positive semi-definite.*

Proof. Same as in Theorem 9 and 10 respectively, simply replacing direct sums with tensor products. □

Appendix B. Kernel Configuration Details

The kernel specification defines the way in which data and knowledge should be treated. The default way of treating compound terms can be declared to be either *sum* or *product*, by writing `compound_kernel(sum)` or `compound_kernel(product)` respectively.

The default atomic kernel is the delta one for symbols, and the product for numbers. Such behavior can be modified by directly specifying the type signature of a given clause or fact. As an example, the following definition overrides the default kernel between `atm` terms in mutagenesis:

```
type(atm(ignore,ignore,cat,cat,num)).
```

It allows to ignore identifiers for molecule and atom, and change the default behavior for atom type (which is a number) to categorical. At this level, it is possible to specify a combining operator for predicate arguments which is different from the default one:

```
type(atm(ignore,ignore,cat,cat,num),product).
```

Here we are stating that atoms of different types will always have zero similarity. Default behaviors can also be overridden by defining specific kernels for particular clauses or facts. This corresponds to specifying distinguished types together to appropriate kernels for them. Thus, the last kernel between atoms could be equivalently specified by writing:

```
term_kernel(atm(_,_,Xa,Xt,Xc), atm(_,_,Ya,Yt,Yc),K) :-
    delta_kernel(Xa,Ya,Ka),
    delta_kernel(Xt,Yt,Kt),
    dot_kernel(Xc,Yc,Kc),
    K is Ka * Kt * Kc.
```

A useful kernel which can be selected is the *functor_equality* kernel as defined in Equation (22). For example, by writing

```
term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
```

at the end of the configuration file it is possible to force the default behavior for all remaining terms to functor equality, where the combination operator employed for internal nodes will be the one specified with the `compound_kernel` statement.

Appendix C. Visitors and Kernels Used in Experiments

C.1 Bongard Problems

```
visit(X):-
    inside(X,A,B),polygon(X,A),polygon(X,B).

compound_kernel(product).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
```

C.2 M-of-N Problems

```
visit(X):-
    string(X,S),substr([A,B],S),comp(A,B).

leaf(substr(_,_)).

compound_kernel(sum).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
```

C.3 Protein Fold Classification

```
visit_global(X):-
    normlen(X,Len),
    normnb_alpha(X,NumAlpha),
    normnb_beta(X,NumBeta).

visit_adjacent(X):-
    adjacent(X,A,B,PosA,TypeA,TypeB),
    normcoil(A,B,LenCoil),
    unit_features(A),
    unit_features(B).

visit_unit(X):-
    sec_struc(X,A),
    unit_features(A).

unit_features(A):-
    normsst(A,B,C,D,E,F,G,H,I,J,K),
    has_pro(A).

unit_features(A):-
    normsst(A,B,C,D,E,F,G,H,I,J,K),
    not(has_pro(A))).


leaf(adjacent(_,_,_,_,_,_)).
leaf(normcoil(_,_,_)).


compound_kernel(sum).

type(normlen(ignore,num)).
type(normnb_alpha(ignore,num)).
type(normnb_beta(ignore,num)).
type(normsst(ignore,ignore,ignore,ignore,ignore,num,ignore,num,num,num,ignore)).
type(adjacent(ignore,ignore,ignore,cat,cat,cat)).
type(normcoil(ignore,ignore,num)).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
```

C.4 Mutagenesis

```
visit_global(X):-
    lumo(X,Lumo),
    logp(X,Logp),
    ind1(X,Ind1),
    inda(X,Inda).

visit_ring_size_5(X):-
    ring_size_5(X,Atoms),
    atoms(X,Atoms).
% ... etc.
```

```
    visit_benzene(X):-
        benzene(X,Atoms),
        atoms(X,Atoms).

    visit_anthracene(X):-
        anthracene(X,[Ring1,Ring2,Ring3]),
        atoms(X,Ring1),
        atoms(X,Ring2),
        atoms(X,Ring3).

    compound_kernel(sum).

type(atm(ignore,ignore,cat,cat,num)).
type(bond(ignore,ignore,ignore,cat)).
type(lumo(ignore,num)).
type(logp(ignore,num)).
type(ind1(ignore,num)).
type(inda(ignore,num)).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
```

```
leaf(benzene(_,_)).
leaf(anthracene(_,_)).
leaf(ring_size_5(_,_)).
% ... etc.

atoms(X,[]).
atoms(X,[H|T]):-
    atm(X,H,_,_,_),atoms(X,T).
```

C.5 Biodegradability

```
 visit_p1(X):-
     sscount(X,_SSType,_SSCount).

 visit_p2(X):-
     p2countnorm(X,_P2Type,_P2Count).
    visit_alcohol(X):-
        alcohol(X,Atoms,Conns),
        atoms(X,Atoms),
        atoms(X,Conns).

    visit_aldehyde(X):-
        aldehyde(X,Atoms,Conns),
        atoms(X,Atoms),
        atoms(X,Conns).


atoms(X,[]).
atoms(X,[H|T]):-
    atm(X,H,_,_,_),
    atoms(X,T).

compound_kernel(sum).

type(atm(ignore,ignore,cat,ignore,ignore)).
type(normlogP(ignore,num)).
type(normmweight(ignore,num)).
type(sscount(ignore,cat,num),product).
type(normp2count(ignore,cat,num),product).

term_kernel(X,Y,K):-
    functor_equality_kernel(X,Y,K).
```

```
visit_global(X):-
    normlogP(X,_LogP),
    normmweight(X,_Mweight).


    visit_ar_halide(X):-
        ar_halide(X,Atoms,Conns),
        atoms(X,Atoms),
        atoms(X,Conns).
    % ... etc.

    leaf(alcohol(_,_,_)).
    leaf(aldehyde(_,_,_)).
    leaf(ar_halide(_,_,_)).
    % ... etc.
```

References

A.W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Transactions on Software Engineering*, 2(3):141–153, 1976.

H. Blockeel and L. De Raedt. Top-down induction of first-order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, 1998.

H. Blockeel, S. Dzeroski, B. Kompare, S. Kramer, B. Pfahringer, and W. Van Laer. Experiments in predicting biodegradability. *Applied Artificial Intelligence*, 18(2):157–181, 2004.

M. Bongard. *Pattern Recognition*. Spartan Books, 1970.

M. Collins and N. Duffy. New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *Proceedings of the Fortieth Annual Meeting on Association for Computational Linguistics*, pages 263–270, Philadelphia, PA, USA, 2002.

C. Cortes, P. Haffner, and M. Mohri. Rational kernels: Theory and algorithms. *Journal of Machine Learning Research*, 5:1035–1062, 2004.

C. Cortes and V.N. Vapnik. Support vector networks. *Machine Learning*, 20:1–25, 1995.

F. Cucker and S. Smale. On the mathematical foundations of learning. *Bulletin (New Series) of the American Mathematical Society*, 39(1):1–49, 2002.

C. M. Cumby and D. Roth. Learning with feature description logics. In S. Matwin and C. Sammut, editors, *Proceedings of the Twelfth International Conference on Inductive Logic Programming*, volume 2583 of *LNAI*, pages 32–47. Springer-Verlag, 2002.

C. M. Cumby and D. Roth. On kernel methods for relational learning. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 107–114, Washington, DC, USA, 2003.

L. De Raedt, K. Kersting, and S. Torge. Towards learning stochastic logic programs from proof-banks. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, pages 752–757, 2005.

P. Frasconi, A. Passerini, S. Muggleton, and H. Lodhi. Declarative kernels. Technical Report RT 2/2004, Dipartimento di Sistemi e Informatica, Università di Firenze, 2004.

Y. Freund and R.E. Schapire. Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296, 1999.

T. Gärtner. A survey of kernels for structured data. *SIGKDD Explorations Newsletter*, 5 (1):49–58, 2003.

T. Gärtner, J.W. Lloyd, and P.A. Flach. Kernels and distances for structured data. *Machine Learning*, 57(3):205–232, 2004.

D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, University of California, Santa Cruz, 1999.

T. Horváth, T. Gärtner, and S. Wrobel. Cyclic pattern kernels for predictive graph mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 158–167. ACM Press, 2004.

T. Horvath, S. Wrobel, and U. Bohnebeck. Relational instance-based learning with lists and terms. *Machine Learning*, 43(1/2):53–80, April 2001.

T. Jaakkola and D. Haussler. Exploiting generative models in discriminative classifiers. In *Advances in Neural Information Processing Systems 11*, pages 487–493, Cambridge, MA, USA, 1999. MIT Press.

A. Karalič and I. Bratko. First order regression. *Machine Learning*, 26(2-3):147–176, 1997.

H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 321–328, Washington, DC, USA, 2003.

G. S. Kimeldorf and G. Wahba. A correspondence between Bayesian estimation on stochastic processes and smoothing by splines. *The Annals of Mathematical Statistics*, 41:495–502, 1970.

A.J. Knobbe, A. Siebes, and B. Marseille. Involving aggregate functions in multi-relational search. In *Proceedings of the Sixth European Conference on Principles and Practice of Knowledge Discovery in Databases*, volume 2431 of *LNCS*, pages 287–298. Springer-Verlag, 2002.

S. Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 812–819, Cambridge/Menlo Park, 1996. AAAI Press/MIT Press.

S. Kramer. *Relational Learning vs. Propositionalization: Investigations in Inductive Logic Programming and Propositional Machine Learning*. PhD thesis, Technischen Universität Wien, Wien, Austria, 1999.

S. Kramer, N. Lavrac, and P. Flach. Propositionalization approaches to relational data mining. In *Relational Data Mining*, pages 262–286. Springer-Verlag, NY, 2000.

T. K. Lakshman and U. S. Reddy. Typed prolog: A semantic reconstruction of the mycroft-O'keefe type system. In V Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–220, San Diego, CA, October 1991. MIT Press.

C. Leslie, E. Eskin, and W.S. Noble. The spectrum kernel: a string kernel for svm protein classification. In *Proceedings of the Seventh Pacific Symposium on Biocomputing*, pages 564–575, Lihue, Hawaii, USA, 2002.

J.W. Lloyd. *Logic for learning: learning comprehensible theories from structured data*. Springer-Verlag, 2003.

H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444, 2002.

P. Mahé, N. Ueda, T. Akutsu, J.-L. Perret, and J.-P. Vert. Extensions of marginalized graph kernels. In R. Greiner and ACM Press D. Schuurmans, editors, *Proceedings of the Twenty-first International Conference on Machine Learning*, pages 552–559, Banff, Alberta, Canada, 2004.

S. Menchetti, F. Costa, and P. Frasconi. Weighted decomposition kernels. In *Proceedings of the Twenty-second International Conference on Machine Learning*, pages 585–592, New York, NY, USA, 2005. ACM Press.

T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation based generalization: a unifying view. *Machine Learning*, 1:47–80, 1986.

T. M. Mitchell, P. E. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine learning: An artificial intelligence approach*, volume 1, pages 163–190. Morgan Kaufmann, 1983.

S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.

S.H. Muggleton, H. Lodhi, A. Amini, and M.J.E. Sternberg. Support vector inductive logic programming. In *Proceedings of the Eighth International Conference on Discovery Science*, volume 3735 of *LNAI*, pages 163–175, 2005.

C. Nédellec, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, volume 32 of *Frontiers in Artificial Intelligence and Applications*, pages 82–103. IOS Press, 1996.

A. Passerini and P. Frasconi. Kernels on prolog ground terms. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 1626–1627, Edinburgh, Scotland, UK, 2005.

C. Perlich and F. Provost. Aggregation-based feature invention and relational concept classes. In *Proceedings of the Ninth SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 167–176. ACM Press, 2003.

T. Poggio and S. Smale. The mathematics of learning: Dealing with data. *Notices of the American Mathematical Society*, 50(5):537–544, 2003.

J. R. Quinlan. Combining instance-based and model-based learning. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 236–243, Amherst, Massachusetts, 1993. Morgan Kaufmann.

J. Ramon and M. Bruynooghe. A framework for defining distances between first-order logic objects. In D. Page, editor, *Proceedings of the Eighth International Conference on Inductive Logic Programming*, volume 1446 of *LNAI*, pages 271–280. Springer-Verlag, 1998.

S. Russell and P. Norvig. *Artifical Intelligence: A Modern Approach*. Prentice-Hall, 2nd edition, 2002.

B. Schölkopf and M.K. Warmuth, editors. *Kernels and Regularization on Graphs*, volume 2777 of *LNCS*, 2003. Springer.

E.Y. Shapiro. *Algorithmic program debugging*. MIT Press, 1983.

J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

A. Srinivasan, S. Muggleton, M. J. E. Sternberg, and R. D. King. Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, 85(1-2):277–299, 1996.

M. Turcotte, S.H. Muggleton, and M.J.E. Sternberg. The effect of relational background knowledge on learning of protein three-dimensional fold signatures. *Machine Learning*, 43(1,2):81–96, April-May 2001.

W. Van de Velde. IDL, or taming the multiplexer. In K. Morik, editor, *Proceedings of the Third European Working Session on Machine Learning*, pages 211–226. Pitmnann, 1989.

F. Van Harmelen and A. Bundy. Explanation based generalization = partial evaluation. *Artificial Intelligence*, 36:401–412, 1988.

V.N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

A.C. Varzi. Parts, wholes, and part-whole relations: the prospects of mereotopology. *Data and Knowledge Engineering*, 20:259–286, 1996.

S.V.N. Viswanathan and A. J. Smola. Fast kernels for string and tree matching. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 569–576. MIT Press, Cambridge, MA, 2003.

Y. Wang and I. Witten. Inducing model trees for continuous classes. In *Proceedings of the Ninth European Conference on Machine Learning*, pages 128–137, Prague, Czech Republic, 1997.

J. M. Zelle and R. J. Mooney. Combining FOIL and EBG to speed-up logic programs. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1106–1111, Chambéry, France, 1993.